Compass: Cost of Migration-aware Placement in Storage Systems

Akshat Verma
IBM India Research Lab
akshatverma@in.ibm.com

Upendra Sharma
IBM India Research Lab
supendra@in.ibm.com

Rohit Jain
IBM India Research Lab
rohitjain@gmail.com

Koustuv Dasgupta
IBM India Research Lab
kdasgupta@in.ibm.com

Abstract— **We investigate methodologies for placement and migration of logical data stores in virtualized storage systems leading to optimum system configuration in a dynamic workload scenario. The aim is to optimize the tradeoff between the performance or operational cost improvement resulting from changes in store placement, and the cost imposed by the involved data migration step. We propose a unified economic utility based framework in which the tradeoff can be formulated as a utility maximization problem where the utility of a configuration is defined as the difference between the benefit of a configuration and the cost of moving to the configuration.**

We present a storage management middleware framework and architecture *Compass* that allows systems designers to plug-in different placement as well as migration techniques for estimation of utilities associated with different configurations. The biggest obstacle in optimizing the placement benefit and migration cost tradeoff is the exponential number of possible configurations that one may have to evaluate. We present algorithms that explore the configuration space efficiently and compute a candidate set of configurations that optimize this cost-benefit tradeoff. Our algorithms have many desirable properties including local optimality. Comprehensive experimental studies demonstrate the efficacy of the proposed framework and exploration algorithms, as our algorithms outperform migration cost-oblivious placement strategies by upto 40% on real OLTP traces for many settings.

I. INTRODUCTION

Storage systems management for guaranteeing application I/O performance is an important and challenging problem being faced by datacenter architects today. For most workloads, disk access latency continues to be several orders of magnitude larger than the computation time. As hard the problem was with directed attached storage, consolidation and virtualization of storage resources in a shared Storage Area Network has made the problem of performance management even harder, since the storage resources are now shared between competing workloads from different applications. In this paper, we propose *Compass*: an integrated performance management architecture and methodology for storage systems that takes advantage of the virtualization layer's capability of migrating data between storage devices transparently. The proposed solution is aimed at handling performance problems in a virtualized storage environment that arise out of factors such as changes in workload intensity and pattern with time. Most existing techniques for solving storage performance problems rely on selective throttling of I/O streams but do not consider change in data placement on storage devices, since the needed data migration is considered to be disruptive. We propose a technique based on reconfiguration involving change in placement of data, where the reconfiguration choices are evaluated for both the potential improvement in performance and the negative impact due to data migration.

The central idea of *Compass* is that by employing frequent but limited local reconfigurations, system can respond to frequent changes in incident workload without large scale disruptions. In contrast, a large scale reconfiguration is very expensive in terms of performance disruption and hence can be undertaken very infrequently, leaving the system unable to take advantages of short term workload variations. We consider an economic utility based framework, in which the utility of a candidate new configuration is evaluated in terms of (a) the cost of data migration on account of the opportunity cost of not serving some requests during migration, and (b) the expected benefit from better service quality in new configuration. A configuration that is *close* to the present configuration and optimizes this trade-off is chosen as the new configuration. The fact that the evaluation is sensitive to the migration cost makes this technique effective. The proposed technique also provides a continuum in performance management between techniques that rely on workload throttling [13], [4], [18] and techniques that rely only on optimizing by placement [11], [2]. Unlike the former, it can solve problems caused by load imbalance to some extent, and by considering the cost of migration it can avoid taking placement decisions that worsen the situation.

The motivation for frequent reconfiguration comes from a number of factors. In a storage service provider setting where a number of workloads are consolidated on a few systems, system reconfiguration could be performed to shift resources between negatively correlated workloads to satisfy the peak requirements of individual workloads, while provisioning resources only for average requirements. The negative correlation could arise from time-zone difference, from the diurnal nature of different workloads, or just from the changing popularity of different data items in a large dataset. A number of studies on I/O workloads have also suggested the self-similar nature of I/O traffic [8], [9]. The implication is that the traffic is bursty over a wide range of time scales. Again, migration cost aware system reconfiguration employed frequently can provide much better performance than coarse grained reconfigurations carried out after long intervals. Used in conjunction with online migration execution techniques such as [5] that minimize performance impact, *Compass* can be a very effective technique.

A. Framework and Contribution

We consider the problem of dynamic resource reconfiguration in storage systems that provides statistical QoS guaran-

tees. QoS guarantees are provided on a request stream defined as an aggregation of I/O requests from an application. The logical aggregation of all data accessed by the requests of a stream is referred to as a store, which is backed up by physical space on storage devices. Typically, QoS guarantees, more formally referred to as service level agreements (SLA), specify an upper bound on the stream request rate and also an obligation on the system to serve a minimum fraction of these requests within a specified latency bound. In this setup, each request is associated with a reward value that the system accrues when the request is serviced within its latency bound. Reconfiguration, accomplished by migrating stores between storage devices, impose a cost on the system since it consumes resources that could otherwise have been used for servicing application requests.

We provide an economic utility based framework using which both the expected benefits of the new configuration and the migration cost can be computed. This provides a basis for evaluating the configuration choices taking into account both the expected benefit and the cost of migration. This is in contrast to store placement techniques that only consider the expected benefit of a configuration regardless of the cost of migration or the expected period during which the workload remains stable. We consider two commonly used placement techniques and show that these techniques perform poorly when used for reconfiguring the system in response to small and short-lived variations in workloads. We propose new algorithms that efficiently explore the configuration space in the neighborhood of configurations given by existing placement techniques to find candidate configurations that optimize the cost benefit tradeoff. We believe *Compass* can be a powerful technique for performance management of storage systems. In our experiments, *Compass* shows significant improvement (up to 40% for a large number of settings) over the approach of using existing placement techniques for use in reconfigurations.

B. Related Work

Much previous work in the area of storage systems management has focused on automated mechanisms for offline near-optimal storage systems design, but only a few of these address the issue of handling dynamic workload changes. Anderson *et al.* proposed Ergastulum [2] for design of storage systems using heuristics based on best-fit bin packing with randomization, to search the large design space. They also proposed Hippodrome [3] for automated storage system configuration that uses an iterative approach to determine storage configuration for a given set of workloads. However, the change in configuration between the iterations is carried out without regard to the adverse impact of migration on system performance.

Other approaches such as Facade[13] and Sleds[4] rely exclusively on throttling workloads to enforce fairness in resource arbitration. Chameleon[18] improves on these approaches by automatically inferring and refining workload and device models at run-time. They are designed to work in scenarios where the total load on the system exceeds the system capacity, but they don't address the problems caused by load imbalance resulting from inefficient placement. Thus, these approaches can be used in conjunction, but not in lieu, of migration based approaches.

Scheuermann *et al.* [17] propose a *disk cooling* mechanism that uses frequent migrations to ensure that the heat (request rate) on all disks is equal while minimizing the total amount of data being moved. Their approach is most similar to our work as they take the cost of migration into account while searching for new placements by considering the size of data being moved. However, their approach is tied strictly to one benefit function (heat or load balancing) and can not be applied to other benefit functions like throughput maximization or response time minimization. Further, since the migration cost is integrated with the placement algorithm, the technique can not be used in conjunction with other placement algorithms. These are exactly the problems that we solve in this work. Our framework is designed to work with any choice of placement algorithm and migration methodology.

The application of reward maximization as a tool in a service provider setting for resource allocation problems has been used by several researchers. Most of these address allocation of resources for various request classes with QoS guarantees so that resources are optimally utilized, thereby maximizing the profit of the providers. Among these, Liu *et al*. [12] proposed a multi-class queuing network model for the resource allocation problem in an offline setting. Verma *et al*. [21] address the problem of admission control for profit maximization of networked service providers. Dasgupta *et al*. [5] use a reward maximization approach for admission control and scheduling of migration requests that compete with application workload. However, none of these formulations are able to capture the tradeoff between the benefit of a new configuration and the cost of migrating to it.

II. COMPASS: MODEL AND ARCHITECTURE

The placement of stores on the disks of a storage subsystem determines the throughput as well as the response time experienced by the application workload. An optimal placement/allocation scheme allocates the stores to the different disks in a manner such that some suitably defined benefit function (e.g, response time averaged over all requests, disk throughput, number of requests that are served within a deadline) is maximized. If the objective of the placement is to minimize a particular metric (e.g. response time), the benefit may be defined as inverse of the metric to transform the minimization version of the placement problem to a (benefit) maximization problem. Hence, the placement problem can always be expressed as a maximization problem with respect to some suitably defined *Benefit* function. The *Benefit* function can similarly be defined for the problem of placing stores on RAID arrays. Although, we talk about the placement problem only in the context of disks in this work, our central idea is applicable to RAID arrays as well.

In real deployment, the optimal placement of stores on disks (one that maximizes benefit) may change with change in workload or storage resources (e.g., addition/deletion of disks) and the stores may have to be periodically reallocated to different disks. In order to achieve the new optimal configuration, some stores are migrated from one disk to another. This migration,

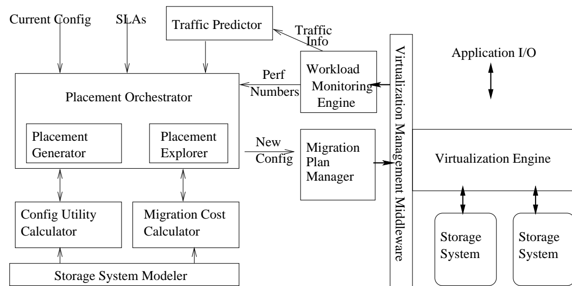


Fig. 1. Compass: An architecture for Performance Management

which may be frequent, imposes additional workload on the storage resources and may lead to degraded performance (and consequently revenue loss) for the application workload. Further, since any particular placement/allocation of stores to disks remains optimal only for the limited period when the workload remains stable, the additional benefit accrued due to the new configuration within that period should outweigh the revenue loss due to the degraded performance of the application workload during the migration phase for the migration to be useful.

Hence, the best placement at any given moment is not the placement that maximizes the benefit but one that optimizes a net utility function capturing this cost-benefit tradeoff, i.e., a placement that has both a high benefit and a low cost of migration from the previous placement. We now formalize the above notion mathematically. Given an initial allocation A_I of N stores to M disks, a benefit function $B(A_i)$ (to denote the benefit accrued per unit time) defined on all possible store allocations $A_i, (i \leq 1 \leq M^N)$, a revenue loss (or migration cost) function $\mathbb{C}(A_i, A_f)$, and a period T for which the new allocation would hold, the profit maximizing allocation is given by

$$\arg \max_{A_i, 1 \leq i \leq M^N} T(B(A_i) - B(A_I)) - \mathbb{C}(A_I, A_i) \qquad (1)$$

i.e., the optimal allocation maximizes the benefit obtained by the new configuration while minimizing the revenue loss due to the impact of migration on the application workload. The goal of this work is to explore such cost-benefit optimizing placements.

A. Framework for evaluating the tradeoff between Benefit and Cost of Configuration

We now present the *Compass* performance management middleware framework that solves the optimization problem posed in Eqn. 1.

The key question while designing an architecture to find the cost-benefit tradeoff optimizing placement is whether to enhance the placement method to become migration cost-aware or to separate the migration cost from the benefit of placement and evaluate the tradeoff in a separate higher-level orchestration component. The advantage of unifying migration with placement methodology is that we directly get the cost-benefit optimizing placement. However, one would have to redesign the placement strategies so that they incorporate the migration cost; which, may or may not be possible for all placement algorithms. Further, since there is no placement algorithm that works best in all workload scenarios, one would have to individually redesign the placement algorithm most suitable for one's setting.

On the other hand, the other design choice of separating migration cost from benefit of placement makes the framework easily extensible, as new placement algorithms or migration methods could be plugged in and used directly. The obstacle in this approach, however, is a method to find candidate cost-benefit optimizing placements as the placement algorithm only provide a single optimal configuration that maximizes benefit oblivious of cost. The only other placement available to us is the previous placement that has zero migration cost. Hence, in this framework, one has to explore the configuration space using these two extreme points (one with zero cost of migration and the other with the maximum benefit).

In *Compass*, we have decided on the latter choice for the *plug-and-play* capability it offers, allowing us to use existing placement methodologies directly. In order to generate candidate cost-benefit optimizing placements, we present algorithms that efficiently explore the configuration space using the two extreme points. The main components of the *Compass* architecture are shown in Fig. 1. The architecture assumes that there is a *Virtualization Engine* between the consumers of storage and the physical devices providing the storage. The *Virtualization Engine* maps the logical view of storage as seen by consumers to the storage provided by physical devices. This indirection affords the *Virtualization Engine* the capability to move data between storage devices while keeping the logical view of data consistent.

A *Workload Monitoring engine* monitors the current workload on the storage subsystem as well as the performance seen by the workload in terms of response time and throughput. The performance measurements are passed to the main control module, which we call the *Placement Orchestrator*. At regular intervals or when SLA violations reach a certain threshold, the *Orchestrator* triggers a configuration evaluation. As a first step towards exploring a new configuration, the *Orchestrator* invokes the *Traffic Predictor* for the predicted workload in the short-term future. The *Traffic Predictor* uses time-series analysis based short-term prediction [10] for estimating the requests arrival rate for each stream. To compute other workload parameters, the *Traffic Predictor* uses a simple history-based model where the weight of a measurement decays exponentially with time.

The *Orchestrator* then obtains the benefit-maximizing placement for this new predicted workload from the *Placement Generator* and invokes the *Placement Explorer* component for any intermediate placements that may optimize the cost-benefit tradeoff. The *Placement Explorer* uses the current benefit-maximizing and cost-minimizing placements (previous placement) as *extreme* points to generate intermediate placements. At this juncture, the *Orchestrator* has a list of eligible placements and uses the *Config Utility Calculator* and *Migration Cost Calculator* to compute the benefit of each placement and the cost of moving to the new placement respectively. The *Config Utility Calculator* and the *Migration Cost Calculator*, in turn, use a *Storage System Modeler* to aid them in their calculation by providing an estimate of the performance numbers for a given workload and placement efficiently. The *Orchestrator* then selects the placement that best optimizes the cost-benefit tradeoff (Eqn. 1). Once a new

configuration is selected, a *Migration Plan Manager* creates and executes a migration plan to migrate the stores according to some migration methodology. The *Virtualization Engine* does the actual task of migration in accordance with the migration plan, thus completing the loop.

Note that both the *Configuration Utility Calculator* and *Migration Cost Calculator* rely on the existence of either an analytical model of the storage devices or a simulator to determine the expected utility. Admittedly, this is a challenging task, but a number of models have been developed by researchers in the past with some degree of success. Analytical models, though hard to construct, have been shown to successfully model disks and disk arrays with controllers [19]. Other models, such as table based lookups with interpolation between table entry configurations [1], that are less accurate but more easily adaptable to device changes have also been proposed and can be used.

B. Store Placement for Benefit Maximization

The benefit of a placement A_h, in a service provider setting, denotes the revenue earned by the provider for serving requests in the placement A_h, where the revenue earned from a request (or set of requests) is calculated based on pre-specified Service Level Agreements (SLA). Common SLAs tend to reward providers for maximizing the number of requests served and minimizing the response time of the served requests. In a single-enterprise setting, the aim of the storage administrator may be to maximize total disk throughput and/or minimize response time without any explicit agreements in place. However, both these settings employ a common notion of optimizing certain objective functions and we capture these objectives with the aid of a general *Benefit* function.

The *Compass* methodology does not depend on the actual benefit function used, which is just an input in finding the placement that optimizes the cost-benefit tradeoff. The choice of Benefit function is dictated rather by the choice of placement algorithms. In this work, we consider the commonly used placement strategies proposed in [11], [2] and accordingly, for an allocation A_h that places N stores on M disks under the assumption that all requests have the same reward, the placement objectives are captured by the following benefit function (Eqn. 2).

$$B(A_h) = \lambda_{A_h} \ \frac{L}{\delta(A_h)} = \sum_{j=1}^{N} \lambda_j \ \frac{L}{\delta_j} \tag{2}$$

where λ_{A_h} is the number of requests served, L is baseline response time, $\delta(A_h)$ is the average response time in the allocation A_h, λ_j is the request rate of stream j and δ_j is the average response time of the requests of stream j in the allocation A_h. The parameter L is a constant and may denote a baseline response time, which could be the target response time for the specific application. In a service provider setting with differentiated rewards r_j for each stream j, based on the negotiated *SLA*, the benefit of served request depends on the reward of the request as well. Hence, we extend the benefit function for a multi-reward setting in the following manner.

$$B(A_h) = \sum_{j=1}^{N} \lambda_j r_j \ \frac{L}{\delta_j} \tag{3}$$

This benefit function captures both facets of most service level agreements: rewarding placements for (i) maximizing throughput and (ii) minimizing response time.

We now discuss some of the common placement strategies used in practice and learn the intuition behind these strategies. The placement of stores on a storage subsystem to optimize some specific objective function has been investigated by many researchers. Lee et al. [11] place files on parallel disks with the aim of minimizing the average response time while ensuring that the load is balanced across the disks. We call this algorithm as the LSV algorithm. Ergastulum strives to find a storage system with the minimum cost and then balances the load across the disks for that storage system. Garg et al. [7] allocate streams to servers in a web farm in order to minimize the average response time of all the stores. Verma et al. [20] solve the same problem for store allocation on parallel disks. Our notion of *Benefit* of a store allocation is able to capture all these diverse objective functions. The benefit of an allocation increases with increase in throughput and decreases with increase in response time and hence, the above benefit metric (Eqn. 3) is rich enough to capture all these settings. To investigate the cost-benefit tradeoff, we will revisit some placement algorithms, namely, the LSV and $Ergastulum$. We will use insights from these algorithms in designing our intermediate selection methodology.

III. Exploring the Configuration Space

The key insight behind the framework presented above is that there might be certain configurations that may not lead to the highest benefit but achieve a better cost-benefit tradeoff, since the migration load to achieve the particular configuration may be very low. We now describe methods to search configurations that optimize this tradeoff.

An obvious way to find such configurations is to perform a *local random search* for allocations near the benefit-maximizing configuration. However, the number of such configurations may be large and exploring all of them may be prohibitively expensive. Hence, instead of a *local random* search, we conduct a more *informed search* using two extreme points; the previous configuration and the new benefit-maximizing configuration. Note that the former represents the cost minimizing placement while the latter represents the benefit maximizing placement. We now describe the insight behind our informed search method that uses the two extreme points.

The reason that the configuration with the highest benefit may not optimize the cost-benefit tradeoff is that the placement methods strive to maximize the benefit oblivious of the earlier configuration and the resultant migration cost. To take an example, LSV sorts the streams based on expected service time ($E(S)$) of the requests of a stream and assign them to the disks in this order such that all disks have equal load. Hence, if the load for a particular stream changes, a disk may not have balanced load and that imbalance may have to be distributed across the disks. Since the disks should maintain a sorted order for $E(S)$, moving to the new allocation would require moving most of the imbalance through all the disks (Fig. 2).

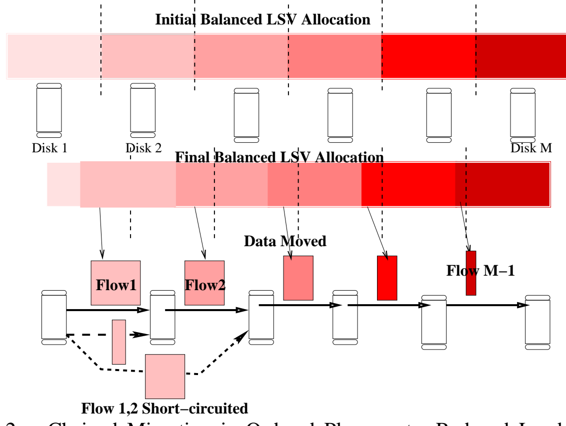


Fig. 2. Chained Migration in Ordered Placements: Reduced Load on one disk leading to a chain of migration involving all disks

One may observe in the above example (Fig. 2) that the total migration load could be $O(M\delta L)$ for an imbalance of δL. To verify, observe that each disk $Disk\ k$ ($k > 1$) receives stores with load of $\frac{\delta L}{M} + \delta L\frac{(M-k)}{M}$ and transfers stores with load of $\delta L\frac{(M-k)}{M}$. Summing up over all the disks, the total data transferred equals $\delta L\frac{M-1}{2}$. We capture this notion of a chain of migrations that arises as a result of load variations, by the term *chained migration*: a migration that requires some stores $S_{i,j}$ placed on disk D_i to be moved to D_j, and some stores $S_{j,k}$ placed on D_j to be moved to D_k. However, if the stores $S_{i,j}$ and $S_{j,k}$ have similar statistical properties, then a transfer of $S_{i,j}$ directly to D_k may lead to a configuration with approximately the same benefit but at a much reduced migration cost. Our basic strategy in selecting candidate placements is to explore placements that are intermediate between the previous and the new benefit-maximizing placement, but do not involve *chained migrations* from the previous placement.

A. Short-circuiting Chained Migrations

The core idea used by us for generating intermediate placements that may optimize the cost benefit tradeoff is, what we call, flow short-circuiting. For a set of stores $S_{i,j}$ and $S_{j,k}$ that are involved in a *chained migration*, flow short-circuiting essentially is replacing the two flows from disk D_i to D_j and D_j to D_k with a single flow from D_i to D_k. In this process, we also need to ensure that the load remains balanced even after this replacement. We name this process as flow short-circuiting since the load that was initially flowing from D_i to D_k via D_j is now directly flowing from D_i to D_k. However, in order to preserve the load-balanced condition, if the total load of $S_{i,j}$ and $S_{j,k}$ are not same, we may not be able to short-circuit the complete flow. In such a scenario, we short-circuit the flow of load equal to the minimum of the loads of $S_{i,j}$ and $S_{j,k}$. In the example of Fig. 2 the flows $Flow1$ and $Flow2$ are short-circuited resulting in a reduction equal to the load generated by $Flow2$, which is the smaller of the two flows, in the total migration load. The same process can be repeated until no chains are left. The details of the short-circuiting algorithm are presented in Fig. 3. In order to preserve the load balanced condition, we compute the total load of $S_{i,j}$ and $S_{j,k}$ stores, and short-circuit a load equal to the minimum of the two, thus ensuring that the load of all three disks D_i, D_j and D_k are preserved across a short-circuit.

function shortCircuitFlow($S_{i,j}, S_{j,k}$)
  Compute the net load inflow IN from D_i *to* D_j
  Compute the net load outflow OUT from D_j *to* D_k
  If (IN < OUT)
    In the outflow of D_i*, change the target disk of stores* $S_{i,j}$ *from* D_j *to* D_k*.*
    Identify a subset $S'_{j,k}$ *of* $S_{j,k}$ *such that load of* S'_{jk} *equals load of* $S_{i,j}$*.*
    In the outflow of D_j*, remove the stores* $S'_{j,k}$*.*
  Else
    In the outflow of D_j*, remove the stores* $S_{j,k}$*.*
    Identify a subset $S'_{i,j}$ *of* $S_{i,j}$ *such that load of* S'_{ij} *equals load of* $S_{j,k}$*.*
    In the outflow of D_i*, change the target disk of stores* $S'_{i,j}$ *from* D_j *to* D_k*.*
end shortCircuitFlow

Fig. 3. Flow Short-Circuiting Algorithm for stores $S_{i,j}$ and $S_{j,k}$

Our intermediate selection methodology essentially consists of (i) computing the migrations required to move from the previous to the new benefit-maximizing placement, (ii) identifying chains among them and (iii) short-circuiting the chains one at a time. We reduce the migration cost in each step by reducing the number of disks that exchange stores (i.e. both receive and transfer stores) and are part of some chained migration. Given an initial migration flow that takes us from an initial to a final allocation, every short-circuit will lead to a migration flow that is one step farther from the final placement and one step closer to the initial allocation. Moreover, every short-circuit reduces the migration cost by removing one chained migration. However, the number of such chained migrations could be large (up to a maximum of ${}^{N}C_3$) and the order in which we short-circuit these chained migrations determines the intermediate states that are selected. We enhance this basic methodology next and specify the order in which these chained migrations are short-circuited and show certain desirable properties of the proposed order for some common placement schemes.

B. Chained Migration Ordering for Sorting Based Placement Algorithms

We now present a method to explore the placement space and find candidate allocations that may optimize the benefit and migration cost tradeoff for placement schemes that sort the streams based on some stream parameter. For ease of elucidation, we consider only the LSV allocation while observing that the same scheme is applicable for other placement schemes that sort streams based on any other stream parameter. The LSV scheme sorts streams based on $E(S)$ of the requests of the stream and assign them to the disks in this order such that all disks have equal load. Hence, if the load for a particular stream changes, the disk on which it is placed may no longer have balanced load and this imbalance may flow across all the disks.

One may note that the LSV scheme derives its performance by isolating streams with large $E(S)$ from streams with small $E(S)$. Hence, while short-circuiting flows, we try to preserve the property of maintaining the sorted order of streams as much as possible, thus isolating the streams with large request

```
function exploreSorted
 for i = 0 to M - 1
    find the chained migration D_i, D_j, D_k such that
       E_{S_{i,j}}(S) - E_{S_{j,k}}(S) is minimized
    shortCircuitFlow(S_{i,j}, S_{j,k})
 end for
end exploreSorted
```

Fig. 4. Candidate Allocation Finding Algorithm for Sorted Order Placements

sizes from those with small request sizes. Hence, in the example of Fig. 2 with $M-2$ chained migrations (and analogously $M-2$ disks that both receive and transfer stores), we select a disk D_j such that $E(S)$ of the stores $S_{j-1,j}$ that are transferred to the disk and $E(S)$ of the stores $S_{j,j+1}$ that are transferred out of the disk are most similar. Hence, after short-circuiting the chained migration, the intermediate placement obtained has the least deviation per unit load short-circuited from the final placement obtained by LSV. Hence, in some sense, this chained migration selection methodology is locally optimal. We describe the details of the selection methodology in Fig. 4.

The algorithm *exploreSorted* computes a set of up to $M-1$ intermediates where each successive intermediate is one more step away from the final placement returned by LSV algorithm. Hence, the set of intermediates provide a sequence of steps that takes us from the new allocation to old allocation, where each step reduces the cost of migration. The number of such intermediates selected is bounded by $\min\{M, N\}$ and hence the method efficiently provides us a small set of intermediates from the exponentially large number of intermediates, one of which may optimize the cost-benefit tradeoff. Moreover, we have the following local optimality result for the intermediates returned by the above algorithm.

Lemma 1: In every iteration of A from k disks exchanges to $k-1$ disk exchanges, the new allocation minimizes the benefit lost per unit load transfer saved amongst all allocations that have $k-1$ disk exchanges.

C. Intermediate Placement Exploration for General Placement Schemes

We now detail out a method to explore the allocation space for intermediate placements when the initial and final placement do not have any sorted total order on the streams assigned to the various disks. Observe that in such a placement scheme, where there is no sorted order between disks, as the heat on one disk increases, the new placement may have disks exchanging streams with more than one disk and the disks exchanging streams may be arbitrarily ordered.

A direct implication of such a scenario is that the number of chained migrations are no longer bounded by M, the number of disks. Instead, in a worst-case scenario, one can verify that the number of chained migrations may be as high as $O(M^3)$. Further, the order in which we short-circuit chained migrations is not clear, as there is no ordering that the placement scheme follows. However, one may note that the response time of a disk depends on the aggregated properties of the streams placed on that disks. Hence, if we can ensure that the aggregated stream parameters placed on a disk before and after the short-circuit are similar, then the intermediate placement obtained by the short-circuit would have a benefit similar to the final placement given by the strategy.

```
function exploreAll
  While there exists at least one Chained Migration
     find the chained migration D_i, D_j, D_k such that
        E_{S_{i,j}}(S) - E_{S_{j,k}}(S) is minimized
     shortCircuitFlow (S_{i,j}, S_{j,k})
  end for
end exploreAll
```

Fig. 5. Candidate Allocation Finding Algorithm for General Placement Schemes

We design the intermediate-exploration algorithm *exploreAll* (Fig. 5) based on these insights. The algorithm takes as input a set of flows that migrate to a final placement from the original placement. It short-circuits the chained migration that may lead to the least variation from the current workload on each disks and terminates when it can not find any chained migration to short-circuit. The following lemma bounds the number of iterations the algorithm may execute.

Lemma 2: The number of intermediates selected by the *exploreAll* algorithm from an initial allocation A_i and a final allocation A_f is bounded by $\min\{N_s, M^2\}$, where N_s is the total number of streams participating in the flow M_{i-f}.

IV. ESTIMATING THE COST ($\mathbb{C}$) OF MIGRATION

We have looked at placements that reduce the cost of migration by reducing the migration load, thus implicitly assuming that the cost of migration is directly related to the migration load. We now formalize the notion of the Cost ($\mathbb{C}$) of a migration more precisely and show how to estimate this cost for some popular migration methodologies.

A. Cost of Migration by Whole Store methodology

This commonly used migration methodology tries to complete the migration as quickly as possible, and is usually referred to as *Whole Store* migration. *Whole Store* migration is not rate-controlled and almost all application requests will miss their QoS requirements while migration is in progress. Hence, the migration of a store of size B_m on a disk that can support a migration throughput of C_m would reject all requests for B_m/C_m time. Let the migration from a configuration A_0 to A_1 be represented as a set $M_{0-1}\{(S_j, D_k, D_l),\}$ where each entry (S_j, D_k, D_l) in M_{0-1} represents migration of a set of stores S_j from disk D_k to D_l. We have the following lemma for the revenue loss due to migration for the *Whole Store* migration methodology.

Lemma 3: The revenue loss due to migration from A_0 to A_1 by the Whole Store migration methodology is given by

$$\sum_{(S_j, D_k, D_l)\in M_{0-1}} B_j/C_k^r \lambda_k \int_{R_o^k}^{\infty} c_r^k p_r^k dr + B_j/C_l^w \lambda_l \int_{R_o^l}^{\infty} c_r^l p_r^l dr \tag{4}$$

where for any given disk D_k, C_k^r and C_k^w are the maximum read throughput and write throughput respectively supported by D_k, c_r^k is the expected capacity used by requests with reward r, p_r^k is the probability that a request has reward r, λ_k is the expected number of requests present at any given time, R_o^k is such that $\lambda_k \int_{R_o^k}^{\infty} c_r^k p_r^k = C_k$, i.e., R_o^k is the reward of the lowest priority request that would have been served if there was no migration.

B. Cost of Migration by QoSMig methodology

Dasgupta et al. [5] propose an adaptive rate-controlled migration methodology QoSMig that optimizes the tradeoff between the migration utility and the impact on client traffic. QoSMig uses the long term forecast to compute a migration rate that is enough to complete the migration within the deadline. Further, it varies the rate of migration adaptively as the client traffic changes: when the client traffic has a large number of high priority requests, migration is throttled below the base-line migration rate and increased when the client traffic has few high priority requests.

The central idea behind the QoSMig methodology is to assign a reward R_m to migration requests that ensures that migration completes within the deadline and at the same time the rate of migration can be increased or decreased as the arrival rate of high reward requests decreases or increases. For the migration of a store of size B_m within a deadline T on a disk with capacity C, R_m is computed as

$$\lambda \int_{R_m}^{\infty} c_r p_r dr \leq C - C_m, \quad (5)$$

where $C_m = B_m/T$, c_r^k is the expected capacity used by requests with reward r and p_r is the probability that a request has reward r. For further details of the methodology and its correctness, we refer the user to [5]. We have the following lemma for revenue loss due to migration when the QoSMig methodology is used for migrating stores.

Lemma 4: The revenue loss due to migration M_{0-1} from an allocation A_0 to A_1 by the QoSMig migration methodology with a deadline T is given by

$$\sum_{(S_j, D_k, D_l) \in M_{0-1}} T\lambda_k \int_0^{R_m^k} c_r^k p_r^k dr + T\lambda_l \int_0^{R_m^l} c_r^l p_r^l dr \quad (6)$$

where for any given disk D_k, c_r^k is the expected capacities used by requests with reward r on D_k, λ^k is the expected number of requests present at any given time, R_m is the reward assigned to migration request by the QoSMig methodology, and p_r^k is the probability that a request of a stream placed on the disk has reward r.

V. Simulation Study

We have conducted a large number of experiments to assess the usefulness of a framework that optimizes the trade-off between the benefit of a new placement and the cost of migration incurred in reaching the new placement. Our experiments also evaluate the effectiveness of our intermediate selection methodologies in exploring the configuration space to reach an allocation that optimizes this tradeoff.

We compare our intermediate selection methodology against the methodologies that either do not take the migration cost or the benefit of a new placement into account. We provide a brief description of these methodologies below.

- Static Placement: In this scheme, the cost of migration is assumed to be large. Hence, at the start of the experiment, an optimal placement is computed based on the forecasted traffic and the same placement continues through the run of the experiment.
- Benefit-based Placement: This is the scheme commonly employed in practice where a new benefit-maximizing placement is computed periodically as the (forecasted) traffic changes. If the benefit of the new allocation is more than the benefit of old allocation, the methodology migrates the data to the new placement.

A. Simulation Setup

Our experimental testbed is modeled along the lines of the *Compass* framework described in Sec. II-A.

For the experimental evaluation, the *Storage System Modeler* component is built on top of the Disksim simulation environment [6]. Disksim has been used in a large number of experimental studies and simulate the behavior of a modern disk very closely. To work in as realistic a setting as possible, we chose the disk model of Seagate Cheetah4LP disk [6] that has been validated by Disksim against the real disk for a wide variety of workloads and simulates its behavior very closely.

In order to study the performance of the various methodologies in a realistic scenario, we used field traces made available by the Storage Performance Council [16]. We used 2 different OLTP traces and identified stores in them. Since the traces used by us have been collected on a disk array that uses disks different from $Cheetah4LP$ disks, as a first step, we scaled the traces so that the disks operate with an average response time close to 100ms, which is a reasonable value for such workload. We then split the original traces into individual traces for each store (each request in the trace contains an application identifier) and fed the traces through our experimental testbed.

	IOPS	Disks	Streams	Mig Deadline	Store Size
Baseline	135	4	28	1200s	0.4GB
Range	110-170	4-16	24-40	800-1800	0.1-0.8

TABLE I
EXPERIMENTAL SETTINGS

As a baseline setting, we used 4 disks and 28 streams for the experiments. The configuration was evaluated every 1 hour with the migration deadline set as 20 minutes. We kept the store size at 10% of the disk size in the baseline setting. We then changed all the above parameters from the baseline, one at a time, to investigate their effect on the performance of the competing methodologies. Table I lists the range of experimental parameters.

B. Results

We first study the behavior of the various algorithms in Fig. 6 as the trace is played with time. We investigate how the Net Utility (defined as Benefit - Cost) of a configuration varies with time for the three competing methodologies when we use LSV as benefit-maximizing placement and QoSMig as the migration methodology. The Benefit is calculated using Eqn. 2 with L as $100ms$, whereas cost is calculated by summing up the expected benefit of all requests that could not be served because of migration. Since we had chosen a good operating point, all methodologies were able to serve all requests and hence, the difference in benefit is only as a result of difference in response time achieved by the methodologies.

The results show that the placement strategy, proposed in *Compass*, is able to explore good intermediates and select

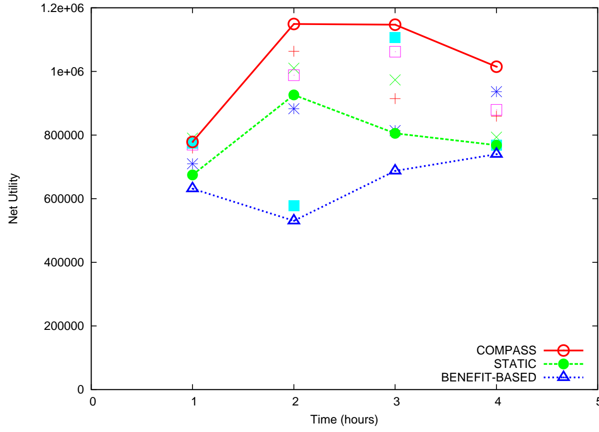
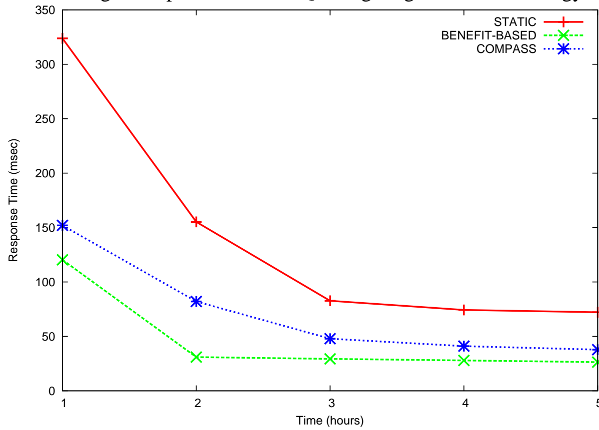
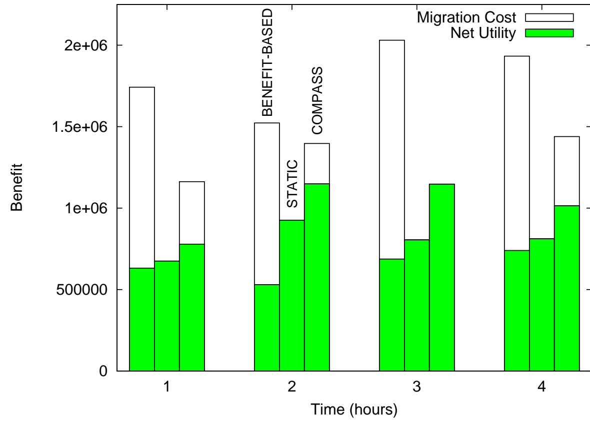


Fig. 6. Benefit of various Placement Strategies and Intermediate allocation with time using LSV placement and QoSMig Migration methodology



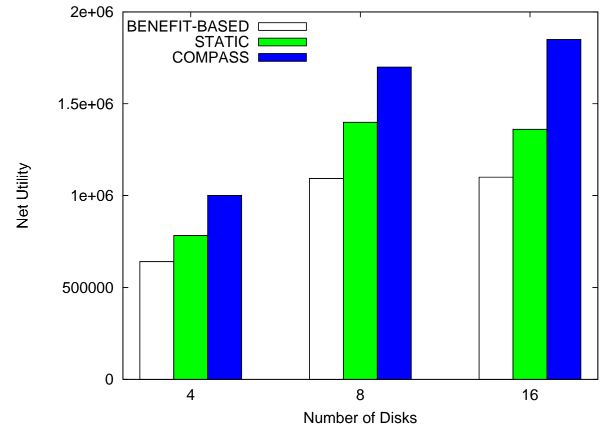
(a)



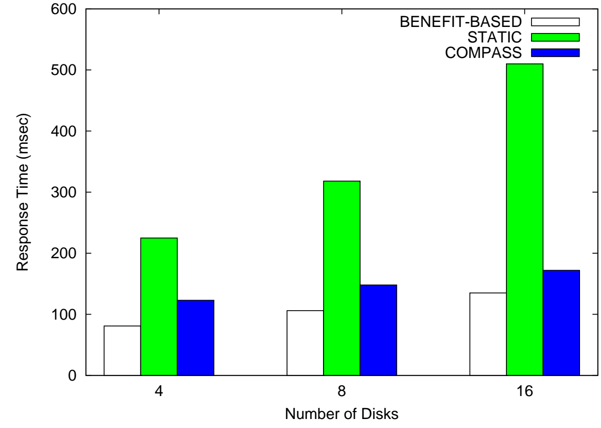
(b)

Fig. 7. (a) Response Time and (b) Migration Cost of different Placement Strategies with time using LSV placement and QoSMig Migration methodology

them appropriately. An interesting observation is that all the intermediates selected by our algorithm (shown as unconnected points in Fig. 6) have Net Utility greater than that of LSV allocation. Further, when the LSV allocation and the initial allocation are very different, we outperform the other algorithms by a more significant margin. This is a result of the fact that when the old and the new allocations are different, the explored intermediates are much varied and show a much wider variation in Net Utility. To understand this behavior better, we look at the response time (Fig. 7(a)) of the allocations selected by the different strategies and observe



(a)



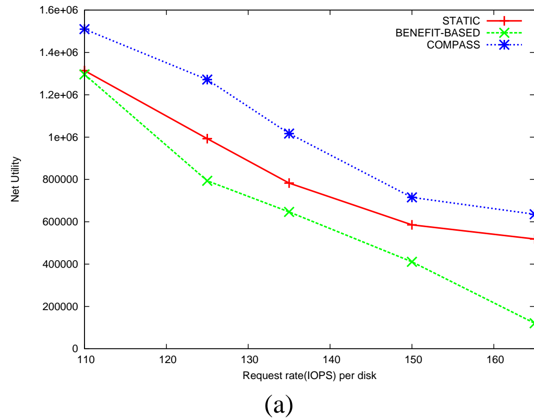
(b)

Fig. 8. (a) Net Utility and (b) Response Time Comparison with increasing number of disks

that *Compass* is able to get a response time close to LSV with very low cost of migration (Fig. 7(b)). On the other hand, *Static* placement has a highly variable response time leading to low benefit whereas *Benefit-based* placement has to incur a high migration cost.

We performed the same experiments with increased number of disks, while compressing the traces, so that the load on the disks remains same. We found that as the number of disks increase, there is a marginal increase in the performance improvement of *Compass* over other methodologies (Fig. 8). This can be attributed to the fact that *Compass* now has more chained migrations to potentially short-circuit and explore a much richer set of intermediates. Hence, as the number of disks increase from 4 to 16, *Compass* outperforms *Static* by a margin of 30%, up from the 20% seen in the 4-disk scenario. For lack of space, for the other sets of experiments, we report our observations only for the 4-disk scenario, while noting that the performance improvement of *Compass* increases with increase in number of disks.

We studied the performance of the various techniques for different combinations of benefit-maximizing placement strategies and migration methodologies with variation in request rate. We observed that LSV as a placement algorithm and $QoSMig$ as a migration methodology are more predictable than $Ergastulum$ placement and $Wholestore$ migration. This is because of the deterministic nature of LSV and the adaptive nature of $QoSMig$. For lack of space, we



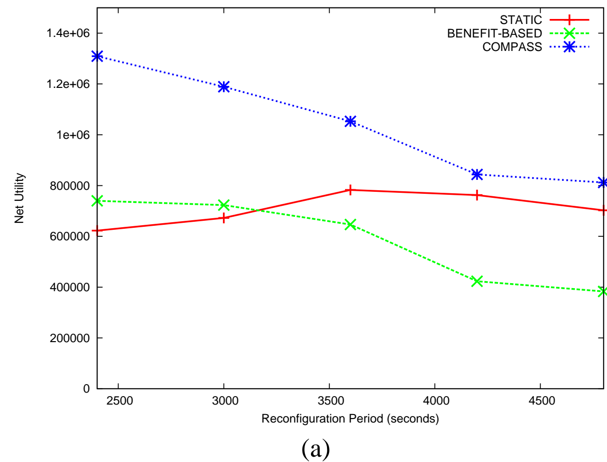
(a)

Fig. 9. Net Utility of different Placement Strategies with Change in Request Rate using LSV allocations and QoSMig

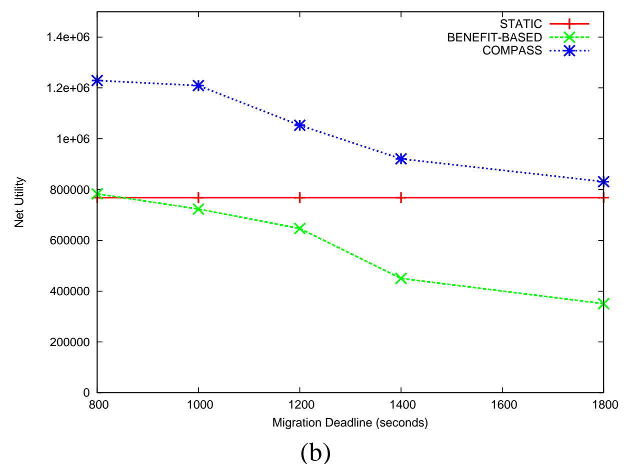
report our results only for this combination. As stated earlier, we vary the request rate by compressing or expanding the trace. An obvious manifestation of the scaling is that the total utility achieved by all methods falls with increase in request rate as the same requests are compressed together leading to an increase in average response time. Fig. 9 shows the performance of the various strategies with LSV as the benefit-maximizing placement algorithm and $QoSMig$ as the migration methodology. The results clearly demonstrate the superiority of *Compass* as it achieves a significantly higher Net Utility as compared to both *Static* placement and the *Benefit-based* (vanilla LSV) placement.

Fig. 10(a) studies the behavior of the various strategies with change in the Reconfiguration period. A large reconfiguration period smoothens out the short-term workload variations and a methodology that does not use frequent migrations may still perform reasonably well. The results validate this intuition as the *Static* placement strategy has the least Net Utility at small reconfiguration period but starts to improve as the Reconfiguration period increases to outperform LSV and even approach our *Compass* methodology. We also study the performance of various algorithms with change in Migration deadline (Fig. 10(b)). As the migration deadline is increased, the duration for which we get any additional benefit due to improved placement is reduced. Hence, both the algorithms that migrate data exhibit a fall in Net Utility with increase in migration deadline whereas the *Static* placement shows no performance change with change in migration deadline.

In Fig 11(a), we study the behavior of the competing strategies as the average number of streams per disk is varied from 6 to 10. One may observe that having a large number of streams per disk leads to a more balanced allocation as fragmentation problems are less pronounced. Further, the requirement of frequent migrations is low as the large number of streams on a disk may smoothen workload variations on the disk. Both these intuitions are validated by our study as the performance of the *Compass* algorithm become similar to the *Static* placement as the number of streams are increased. However, as the number of streams increase, the number of variations in the sorted order of streams with time increases quadratically. Hence, a cost-oblivious algorithm may resort to large scale migrations, which may improve the benefit only marginally. Hence, the Net Utility of the $Benefit$-$based$ algorithm falls with increase



(a)



(b)

Fig. 10. Net Utility of different Placement Strategies with change in (a) Reconfi guration Period and (b) Migration deadline for LSV allocation and QoSMig Migration methodology

in the number of streams.

We have found in most of our experiments that *Static* placement outperforms the *Benefit-based* placement, and a possible reason for that could be that we have stores with very low temperatures (temperature is defined as the request rate to the store per unit space used). Hence, we now vary the size (space used) of each store and study the behavior (Fig. 11 (b)). It is natural to expect that as the store size decreases, the cost of migration decreases and, as a result, both *Benefit-based* and *Compass* methods have to to pay a lower migration cost and hence, their performance improves with decrease in store size. On the other hand, the Net Utility of *Static* should not change with variation in store size. We found both these intuitions to be validated by our results (Fig. 11(b)).

Our experiments conclusively establish the superiority of *Compass* over existing methodologies under a wide variety of workload settings. *Compass* is especially effective for mid-sized stores under moderate to heavy load.

VI. Discussion

We have presented *Compass*, a methodology for performance management of storage systems that optimizes the tradeoff between the cost of migration and the expected improvement in performance as a result of the configuration resulting from migration. This central idea has been shown to be effective in addressing performance problems resulting from load imbalance between various storage subsystems. We

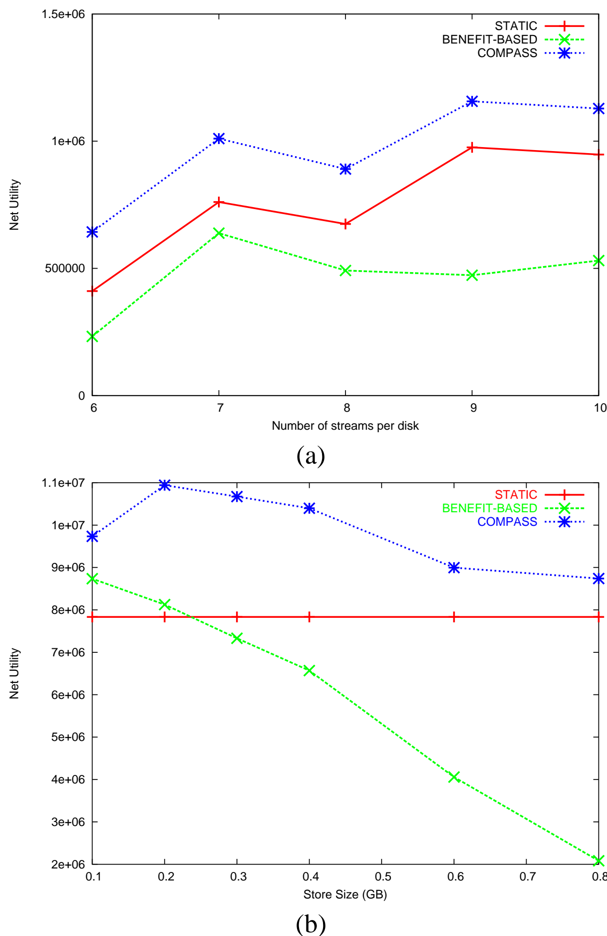


Fig. 11. Net Utility of different Placement Strategies with change in (a) Streams per disk and (b) Store size for LSV allocation and QoSMig Migration methodology

have also presented algorithms for efficient search of configurations, which optimize the tradeoff, in the neighborhood around the configurations given by placement strategies. *Compass* is aimed at handling performance problems in medium time frame (on the order of couple of hours) resulting from workload variations, that are not addressed by load throttling based techniques that work on much shorter time frame.

We now discuss the computational overhead of using $Compass$ as opposed to using a benefit-based placement algorithm like LSV directly. For a benefit-based (or Vanilla) placement strategy, that does not take into account the cost of migration, the computational overhead of the placement strategy is bounded by the running time of the placement algorithm (T_A). In $Compass$, we additionally explore the configuration space for intermediates, and for each of these intermediates, the benefit of the intermediate and the cost of moving to the intermediate is computed. These computations are based on mathematical models of the underlying disk subsystem (we use Disksim in our experimental study) and hence, very fast as compared to the duration of actual migrations. Further, even a vanilla placement strategy also uses the same models for generating its new placement and hence would suffer if the disk model is very detailed and benefit computation takes a long time.

Hence, the only pitfall that the $Compass$ strategy may suffer from is evaluating a large number of intermediates. However, as we have shown, (Lemma 2), the number of such intermediates is bounded by the number of disks M for sorting based placement strategies and M^2 for any general placement strategy. Combining this with the fact that data migration is a very time consuming operation as opposed to computation of simple mathematical functions, the overhead of $Compass$ turns out to be insignificant and transparent to the user in our experiments.

Our work opens up many promising areas to explore. Avenues for future work include allowing migration at granularities smaller than the whole store. Current virtualization technology allows migration at the level of individual RAID stripe, but the interplay of factors such as request locality that can affect performance needs to be carefully evaluated. We also want to evaluate the efficacy of the proposed approach when used with other placement algorithms.

References

[1] E. Anderson. Simple Table-based Modeling of Storage Devices. HP Laboratories SSP Technical Report HPL-SSP-2001-4, 2001.

[2] E. Anderson, M. Kallahalla, S. Spence, R. Swaminathan, and Q. Wang. Ergastulum: An Approach to Solving the Workload and Device Confi guration Problem. HP Labs technical memo HPL-SSP-2001-05, 2001.

[3] E. Anderson, M. Hobbs, K. Keeton, and S. Spence. Hippodrome: Running Circles Around Storage Administration. In *USENIX FAST 2002*

[4] D.D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. P. Lee. Performance Virtualization for Large-Scale Storage Systems. In *Proc. of 22nd SRDS*, 2003.

[5] K. Dasgupta, R. Ghosal, R. Jain, U. Sharma, and A. Verma. QoSMig: Adaptive Rate-Controlled Migration of Bulk Data in Storage Systems. In *Proc. IEEE ICDE*, 2005.

[6] DiskSim Simulation Environment, at *http://www.pdl.cmu.edu/Disksim/*.

[7] R. Garg, P.N. Shahabuddin, and A. Verma. Optimal Assignment of Streams in Server Farms. In *IBM Technical Report*, 2003.

[8] S.D. Gribble, G.S. Manku, D. Rosseli, E.A. Brewer, T.J. Gibson, and E. L. Miller . Self-Similarity in File-System Traffi c. In *ACM SIGMETRICS 1998*, p 141-1 50.

[9] W. Hsu, and A. J. Smith. Characteristics of IO Traffi c in personal Computer and Server Workloads. In *IBM Systems Journal*, 42(2), 2003.

[10] A. K. Iyengar, M. S. Squillante, and L. Zhang. Analysis and characterization of large-scale Web Server Access Patterns and Performance. In *Proc. ACM World Wide Web Conference*, 1999.

[11] L.W. Lee, P. Scheuermann, and R. Vingralek. File Assignment in Parallel I/O Systems with Minimal Variance of Service Time. In *IEEE Transactions on Computers* 49(2), 2000.

[12] Z. Liu, M.S. Squillante, and J.L. Wolf. On Maximizing Ser vice-Level Agreement Profi ts. In *Proc. ACM Conf. on Electronic Commerce*, 2001.

[13] C. R. Lumb, A. Merchant, G. A. Alvarez. Facade: virtual storage device with performance guarantees. In *Proc. of USENIX FAST* 2003.

[14] IBM SAN Volume Controller at *http://www.ibm.com/storage*.

[15] P. Marbach, and R. Berry. Downlink Resource Allocation and Pricing for Wireless Networks. In *Proc. IEEE Infocom*, 2002.

[16] Storage Trace Repository, at *http://traces.cs.umass.edu/storage/*.

[17] P. Scheuermann, G. Weikum, and P. Zabback. Data partitioning and load balancing in parallel disk systems. In *VLDB Journal: Very Large Data Bases*, 7(1):4866, 1998.

[18] S. Uttamchandani, L. Yin, G. A. Alvarez, J. Palmer, and G. Agha. CHAMELEON: A Self-Evolving, Fully-Adaptive Resource Arbitrator for Storage Systems". In *USENIX Annual Technical Conference*, 2005.

[19] E. Varki, A. Merchant, J. Xu, and X. Qiu. Issues and Challenges in the Performan ce Analysis of Real Disk Arrays. In *IEEE Transactions on Parallel and Distr ibuted Systems(TPDS)* 15(6):559-574, June 2004.

[20] A. Verma and A. Anand. On Store Placement for Response Time Minimization in Parallel Disks. In *IEEE ICDCS*, 2006.

[21] A. Verma, and S. Ghosal. Admission Control for Profi t Maximization of Networked Service Providers. In *Proc. Int'l World Wide Web Conference*, 2003.